Using a Recurrent Neural Network to Make Stock Price Predictions *

Antonio Mendoza
NCSSM Online
North Carolina School of Science and Mathematics
Durham, North Carolina

January 9, 2018

Abstract: The goal of this research paper is to experiment with how well a basic Recurrent Neural Network with Long-Short-Term-Memory cells can predict the daily adjusted daily closing prices for the company Tesla in December of 2016. First, I retrieved a full year of historical financial data using the Yahoo finance api. Next, I chose which features I would like to use in my model to make a price prediction. Then, I scaled and formatted my data to feed through a Recurrent Neural Network that I programmed using the TensorFlow library in python. Finally, I plotted the network's predictions and other important indicators to gauge how well my network performed with the financial data I trained it with. Ultimately, my RNN accurately predicted the overall price-direction for a 36 day time period and roughly modeled the daily adjusted close prices to a point where there were distinct similarities between the predictions and actual prices. However, this model was crude at predicting daily price directions. Investing comes with big risks, but for the first time in history, advanced models such as Neural Networks have shown serious potential for modeling future stock prices, so it is important to test how viable this solution really is before adoption.

Key words: Machine Learning, Recurrent Neural Networks, Long-Short-Term-Memory Cells, Stock Predictions, Tesla, Mean Squared Error, Time-Series Data

Introduction

Financial exchanges have had a very deep-rooted history dating back all the way to the 13th century. Back then, Venice, a very powerful trading nation, relied heavily on gold and silver to maintain its economy. However due to the constant inflow and outflow of these metals, Venetian traders created a system of exchange currencies which were used to extrinsically hold the values of gold and silver to maintain high

*Correspondence to: mendoza18a@ncssm.edu

levels of trade throughout their influence. The prices of these exchange currencies changed, so some would lose or gain value by holding these currencies [1].

This is much similar to the stock market in the modern world. Companies will sell shares, which are pieces of the company, when they want to raise more capital for some kind of objective. A share is synonymous to the words 'stock' and 'equity'. By holding a share, one has a claim on the company's assets and earnings. The more shares one has for a specific company means that one will also have a larger claim on the assets and earnings for that company [2].

So, as the value of a business rises or falls, so does the value of its share. Thus, investors who buy shares belonging to a company must always be weary of the future valuation of that business. Or in other words, whether or not the value of their stock will rise or fall.

Knowing when to make a good investment in the stock market is the most crucial judgement any investor can make. Unfortunately, it is very difficult to predict when the value of a company will rise or fall due to a variety of factors. These factors can be as simple as previous price action or as complicated as deeply knowing the individuals which are responsible for an organization.

An experienced investor will familiarize themselves with many technical analysis techniques and will acquaint themselves tremendously well with the company they decide to invest in.

Regardless of experience however, it is impossible to predict the future. Virtually everyone will lose and win in the market at some point. In fact, Burton Malkiel, author and Princeton professor, said that, "A blindfolded monkey throwing darts at a newspaper's financial pages could select a portfolio that would do just as well as one carefully selected by experts" [3].

As gloomy as this statement is, many investment firms and banks such as J.P. Morgan and Co and Bank of America are still hiring quantitative analysts for detecting financial patterns in data. So the question becomes what makes this the case.

With computational power to process and analyze big data increasing exponentially, new state of the art models have been being applied to discover new trends and patterns in data. This is where machine learning comes in. Machine learning models attempt to automatically find patterns in data and then predict future outcomes. By automatically, rather than manually, detecting data patterns [4], huge amounts of computational power are used. So applying these machine learning models have only recently become possible.

One of these such models is known as a Neural Network which has actually been around since 1943 when neurophysiologist Warren McCulloch and mathematician Walter Pitts published a paper about how neurons may work and attempted to design a model based off of their understanding. However, it was not until much later that researchers would start being able to be implement this model effectively [5] .

As mentioned above, a Neural Network is a biologically-inspired mathematical model which simulates how neurons fire and the connections between them strengthen. The main idea of a Neural Network begins with 'training data' which is X data with Y Labels. For example, training data could consist of a hundred pictures, each with the label *"there is/isn't a cat*

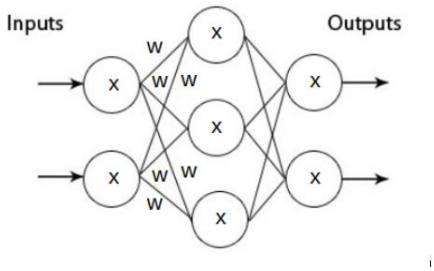

Figure 1: MLP: A Simple three-layered Neural Network

in this image'. The training data will then be fed through the network which will attempt to decipher whether or not a picture contains a cat or not. Upon each iteration of feeding the training data through the network, the model will update itself based on the previous prediction it made by trying to minimize its prediction error with the dependent *Y* data labels. This makes the model more 'intelligent' and increases the accuracy of its predictions over time. Once the model has achieved a consistent and precise accuracy, it can be saved so that unlabeled data can be fed through the the network at anytime and predict whether any image contains a cat or not.

Neural Networks can come in many shapes and sizes, but perhaps the simplest form of this model is what is known as a multilayer perceptron (MLP) as shown in **Figure** 1. A MLP is a Neural Network that only has one hidden layer. The detailed explanation below describes how an image of a cat may cycle through a MLP.

1. Each of the inputs in the first layer *(x)* for each node *(j)* in **Figure** 1 will be a gray-scale pixel value between the range of zero and one.
    - The total inputs, in this case two, will form a layer of inputs which would be a vector, or a tensor. A tensor is any n-dimensional vector. (Besides the inputs, each node in the network represents a single neuron).
2. This input tensor will then be multiplied by a tensor of randomized weights *(w)* between the input and first layer which are shown as straight lines in **Figure** 1. This multiplication results in a vector which describes the next layer of nodes in the network.
    - Each weight is a numerical value which describes the relative importance of each node. By having weights, the networks is able to make decisions based on the size or numerical value of each weight.
3. Afterwards, a bias *(b)* is typically added to the sum of the input and weight products from all of the nodes in the previous layer for a specific node in the next layer, $\sum_j w_j x_j$, so that there will not be any nodes with a value of zero (which will be explained in the next step).
4. The layer of neurons after the input layer

is also known as the 'hidden layer'. For each neuron in the hidden layer, the sum of the neuron and weight products from all of the nodes in the previous layer, $\sum_j w_j x_j + b$, are squished with an activation function to add an element of nonlinearity to the model.

- Without an activation function, the neuron would simply 'fire' or 'not fire' which would make the network linear. However, in the presence of an activation function, it will give each neuron a 'firing strength' which will be a numerical value typically between 0 and 1, making the network nonlinear.
- Popular activation functions are shown in Equations 1, 2, and 3. These are the Sigmoid, Tanh, and Rectified Linear Unit functions respectively.

$$s(x) = \frac{e^x}{e^x + 1} \tag{1}$$

$$tanh(x) = 2s(2x) - 1 \tag{2}$$

$$f(x) = x^+ = max(0, x) \tag{3}$$

- If the value of the neuron is zero, it will not fire at all, so that is why an arbitrary bias value is added to prevent the neuron from not firing at all, initially [6].

5. The process of steps two through four are repeated depending on the number of hidden layers in the network until the output layer (a) is reached.
6. For training, the output vector (Values between 0 and 1 in this example) is compared to the labels of the input data which have the same tensor dimensionality. So In this example, the labels would also be binary (0 or 1) for if there is or isn't a cat in an image. Each time the data is trained, the weights between the hidden layers become updated through a process known as backpropagation, thus causing the output values to change.
7. An epoch is known as one cycle of data through the network; a cycle would be each time data is being fed forward and then back-propagated to update the weights in the network. A Neural Network will be trained with however many epochs it takes to allow the network to make accurate predictions; in this example, until it can predict if there is a cat in an image.

There are two functions used for training. One calculates the cost of the output layer, or in other words, the total error between the output layer and labeled input data. The other function minimizes this error using a calculus process known as gradient descent. There are many kinds of cost functions, each with a purpose, but a general example of a cost function is shown in equation 4, which is also known as the mean squared error (MSE). Where *(w)* represents the weights, *(b)* the bias, *(x)* the inputs, *(n)* the number of training inputs, and *(a)* the outputs.

$$C(w, b) \equiv 2n \sum_x ||y(x) - a||^2. \tag{4}$$

The ultimate goal for this cost function is to converge to zero, so that the output from the model

will match the labeled input data. This is possible by manipulating the weight and bias values in the network. As mentioned above, the weight and bias values are adjusted with an algorithm known as gradient descent to minimize the cost, which is also known as the optimizer function.

Since the cost function can be multivariate (sometimes up to the millions of variables), the gradient descent algorithm must attempt to find local minimum on a multi-dimensional function; traditional calculus optimizations simply become unfeasible with millions of variables. The gradient descent algorithm, shown in Equation 5 will pick a random starting point for each of the cost function's variables and will move in whichever direction that lowers the cost.

$$\triangle C \approx \frac{\partial C}{\partial v1} \triangle v1 + \frac{\partial C}{\partial v2} \triangle v2 \qquad (5)$$

$\triangle v1$ and $\triangle v2$ are vectors that represent changes in the cost function's input variables which can also be described as changes of direction. One may have as many *'v'* variables as the network requires; in this example there are only two. These 'directions' will change so that $\triangle C$ becomes negative, reducing the cost. This function can be explained a lot more in depth within Reference 7 due to its complexity consisting of transpose operations, partial derivatives, gradient vectors, and the learning rate parameter.

Reference 7 also continues to describe how to apply the gradient descent algorithm to update the weights and biases, and how using an alternative optimizer algorithm, stochastic gradient descent, and smaller chunks of data at a time is more efficient than just gradient descent [7].

Neural Networks are incredibly powerful at detecting very subtle patterns and are great at incorporating logic. They can achieve much greater complexities than the MLP described above. Many parameters like choosing the number of neurons, hidden layers, weights, biases, outputs, functions, and features all affect the performance of the Neural Network. It takes serious amounts of time to perfect and develop a truly remarkable Neural Network.

This is why investment firms and banks are starting to take notice to these very powerful models. Quite literally on the J.P. Morgan and Co quantitative analysis careers page, testing and perfecting 'theoretical concepts' for making intelligent investments is part of the job description. This is indicative that they are using some kind of machine learning model to help them make financial predictions. Additionally some companies may have potentially already reaped the benefits from powerful machine learning models. The company MJ Futures has outrageously claimed over 199.2 percent returns over a period of two years on their investments by using a Neural Network (They have yet provide insight to the technicalities of their model). [5].

However, one does not need to be an expert or experienced data scientist to make machine learning models now. Google's TensorFlow library for python has made it extraordinarily easy to create effective and high-accuracy Neural Networks. That is why in this paper, a Recurrent Neural Network with Long-Short-Term-Memory (LSTM) neurons are used to attempt to predict stock prices using Google's TensorFlow module in Python.

The reason why a Recurrent Neural Network (RNN) is used instead of a a traditional deep-learning network is because RNNs are able to

take into account time-series data. For example, consider these two sentences, “My father’s name is David. David is my father”. A traditional Neural Network would not be able to predict this last word since it would need to take into account that David is my father in the first sentence, but since RNNS account for time series data, it would easily be able to predict the last word in the second sentence. But instead of working with sentences, the model in this paper will be deriving patterns from past price action. RNNs with LSTM cells can accomplish these feats by having its neurons loop their prediction tensor outputs back into itself as-well as having them link up to to other neurons through a series of ‘gates’ in each neuron [8]. This complex concept is explained in much greater depth within reference 8.

The goal of this research paper is to experiment with how well a basic RNN with LSTM cells can predict the daily adjusted daily closing prices for the company Tesla in December of 2016.

Computational Approach

Every computational step that I took in this paper utilized the programming language python. Please refer to the scripts in the supplemental code sections for any questions or concerns.

The first step that I took to make my stock-predicting RNN was to get the data. To do this, I used the yahoo finance api to call in financial data from a company of my choice, Tesla, abbreviated to TSLA. Additionally, I chose the start and end dates for an entire year of financial data that I wanted to use in my model. The starting date that I chose was January 1st, 2016, and the end date I chose was December 31st, 2016. In total, I retrieved 252 days of financial data.

After retrieving the data, I downloaded it in a .csv format so that I could better visualize the contents of the financial data. This .csv had six columns; the headers are shown below:

”Date”, ”Open”, ”Close”, ”Low”, ”High”, ”Adjusted Close”

Date refers to the date of the financial records, open refers to the opening stock price for a specific date, close refers to the closing stock price for a specific data, low refers to the lowest stock price for any given data, high refers to the highest stock price for any given date, and Adjusted Close refers to to the closing stock price for any given day after being amended for any distribution or corporation changes to that price. The code for this script can be found in the supplemental code pt.1 section.

The next step I took was to parse the financial data. I did this by reading in the TSLA financial csv data from another python script and by separating each column of data into a variable. Then, I plotted the yearly stock price by using dates as the index and the adjusted close on the y-axis with the python module matplotlib. This result can be seen in **Figure** 2. The code for this script can be found in the supplemental code pt.2 section.

My next steps were to choose which data to use for my RNN and then decide on how to format that data. After some initial brainstorming, I decided to predict the adjusted closing price given a specific date of financial data. Now knowing that I would be using the date and adjusted close columns of data, I would have to transform them into suitable formats for the RNN. To do this,

I first substituted each date with a list of consecutively increasing numbers, so a range of 1 to 252. Then, I appended this range to the adjusted close list and used skylearn's minimum-maximum scaler to transform all of these values between 0-1. I did this because Neural Networks have been shown to perform better when all of its input-output data are on the same scale. Once I scaled my data, I parsed out each the scaled dates and prices into a unique variable again. Afterwards, I had to decide which portion I wanted use as 'training data' and which portion I wanted to use as 'testing data' (testing data meaning the dates that I wanted to predict the prices for) . Almost every time, one will want their training portion to be larger than their testing portion, so I used the first 216 dates as training data and the last 36 as testing data. Since I decided to us an RNN, I would have to turn my lists of data into a a 3D tensor because that is the only tensor shape that the TensorFlow LSTM cell is compatible with. More specifically, TensorFlow requires an input tensor the shape of the batch size by the number of sequences in each batch by the time steps in each sequence for the inputs. Batch size refers to the size of the data chunks that the training data can be divided into, the number of sequences refers to the number of sequences in each batch, and the time step refers to the number of time points within one sequence in each batch. Arbitrarily, I decided to break the training data into six batches, containing six sequences with a time step of six. This resulted in a tensor the shape of $6 * 6 * 6$. This tensor shape can easily be shaped back into the original list of 216 training data points (Just multiply $6 * 6 * 6$; I got lucky with this one). Each batch of training data, a tensor the shape of $6 * 6$, has corresponding training data labels: the prices. I broke the training data labels down into six batches, however, instead of having a $6 * 6$ tensor for each batch, the training labels have a tensor the shape of $1 * 36$. This means that for each prediction the Neural Network will make, the output will simply be a sequence of 36 predicted prices. The goal of doing this is to plot these 36 predicted prices and compare them to the actual last 36 prices of the TSLA stock data. To recap, I started with 252 days of financial data, I used the first 216 to train my network and the last 36 to compare my prediction to. Finally, I plotted the scaled testing data prices as can be seen in **Figure** 3; the prediction will be compared to this plot The code for this script can be found in the supplemental code pt.3 section.

The last python script I wrote was the RNN with TensorFlow (Please note that for any technical questions, refer to the script). Every parameter for the network such as the number of epochs and the RNN size were decided completely arbitrarily. The way that TensorFlow functions is by first defining a computational graph, in my case the RNN. X and Y placeholder variables, which substitute the training data initially, are used in the computational graph. I defined the graph in this script as a function. The next step for running a Neural Network with TensorFlow is to define the TensorFlow session. I do this by creating another function that I can execute to run the session. In this session, I initialize all of my TensorFlow variables, and assign my training data into the TensorFlow placement holder variables. Since I set the number of epochs for this network to 50, the training data will iterate through network will 50 times in the TensorFlow session. For each epoch, the six batches of training data are compared with the price labels for their corresponding dates, and the cost is calculated. Once the cost is calculated for each batch, it is optimized with TensorFlow's Adam

optimizer which utilizes stochastic gradient descent. For each epoch, a scatter plot, which plots the predicted price against the actual price, and a price comparison plot are generated. At the end of training, the mean squared error (calculated by the cost function) and price direction accuracy are plotted for all epochs. The code for this script can be found in the supplemental code pt.4 section.

Results and Discussion

To start, **Figure** 2 shows the entire past years stock prices for the company TSLA. At the beginning of the year it seems as though TSLA took a big dip, but then for the rest of the year it had been oscillating up and downwards. I show this Figure so that one can become familiar with how the recent stock values for TSLA had fluctuated.

For the last 36 days, about a month, shown in **Figure** 2, are the time points in which I decided to make the adjusted close predictions with only the dates. As mentioned in the procedure, In order for decent results from my RNN, I needed to scale the prices and dates with which I would be testing with. This can be seen in **Figure** 3. Despite both the X and Y axes being on a scale from 0 to 1, this graph would have looked identical to one where I may have used 'un-scaled' dates and prices. Additionally, the scaled data can easily be converted back into the real prices and dates. Once the RNN returns a prediction, I will overlay that prediction on **Figure** 3 which shows the actual adjusted closing price for a 36 day time period.

As also mentioned in the procedure, for each epoch, a scatter plot and price plot were generated for comparing actual prices to prediction data. **Figure** 4 and 7 show the price plot and scatter plot after one epoch of training respectively. I generated both of these same kinds of graphs for epoch 30 (**Figure**s 5 and 8) and 50 (**Figure**s 6 and 9) to show the gradual change of my RNN's predictions.

To gauge the overall performance of the RNN, I plotted the mean squared error to observe any trends that might occur which can be seen in **Figure** 10. To gauge accuracy, I compared the prediction to the actual prices and looked to see the same price action on a day to day basis between them (The more days between the prediction and actual price that showed the same directional movement would make the prediction more correct). This price-direction accuracy graph can be seen in **Figure** 11.

Conclusions

Clearly after the first epoch, my network's stock prediction almost looked completely random. In **Figure** 4, the prediction was so incorrect that it made the actual prices look like a straight line in comparison. However, this is expected because the network hasn't had a long enough time to train to make an accurate prediction. A similar result can be seen in **Figure** 7. This figure was made to detect whether or not there is some kind of correlation between the actual price and prediction, but this figure looks more like a plot of randoms dots and shows no correlation between the two variables.

After 30 epochs, the price prediction results had changed drastically. **Figure** 5 shows phenomenal similarity between the prediction prices and the actual prices. For example, between November 19th and December 3rd, both prices show an increase, level out, and then decrease again together. Not only that, both the prediction and actual price continue to gradually increase together from December 3rd to December 24th. Continuing on to **Figure** 8, There is a very dis-

tinct positive correlation between the prediction prices and the actual prices. This is a very good sign that my model is modeling TSLA's 'future' stock prices accurately.

After 50 epochs, the prediction results did not seem to get any more accurate. They actually seemed to be over-fitting the actual prices and the previous predictions. **Figure** 6 shows that despite some of the prediction results getting closer to the actual prices, the first two weeks of the prediction seemed to be dropping off of the chart.**Figure** 9 also shows almost identical results to **Figure** 8, the same positive correlation. So despite the network's prediction still having a positive correlation, the prediction was deviating away form the actual price's general shape. These results indicated that I shouldn't train the network after 30 epochs because the predictions gradually started to over-fit and become more unusable.

Overall, the error in the network's predictions reduced drastically after the first 10 epochs. **Figure** 10 shows the error in the network's predictions converging close to zero which is really good. This means that the values of the predictions were getting closer to the actual price values. Since the prediction values started to fit the actual prices, this makes my model usable for future price predictions.

However, the most interesting take away from all of these results can be seen evidently in **Figure** 11. This graph shows the price-wise accuracy percentage for each prediction of each epoch. Unlike the phenomenal results of **Figure** 10, the price-wise accuracy percentage stayed strangely stagnant. Throughout the entirety of training, this accuracy percentage stayed close to around an average of 50 percent accuracy. The price can only go down and up, so there are only two predictions my network can make. Since my RNN only predicted the correct price action about half of the time, this suggests that these results are very randomized and insignificant.

What is so strange about this is that the network's predictions started to converge and get closer to the actual prices, but failed miserably at making accurate daily price direction changes.

In conclusion, my RNN accurately predicted the overall price-direction for a 36 day time period and roughly modeled the daily stock prices to a point where there were distinct similarities between the predictions and actual prices. However, this model was crude at predicting daily price directions.

Some explanations for why this is the case are as follows:

- Neural Networks typically have millions of training data points, and I only used 216. This is likely to be the reason my prediction started over-fitting the actual prices. Using more data may greatly increase the accuracy of this network.
- Neural Networks almost always use more than one feature for predictions. By only using one feature (the current date), I limited the accuracy of my network In the future, I plan on researching more appropriate features which are indicative of price to use for my model.
- I chose all of the parameters for my RNN at random. Carefully selecting variables like the size of my RNN may greatly increase the accuracy of the network.

As mentioned in the introduction, it takes a

tremendous amount of dedication and time to make a truly well performing Neural Network. I feel like, although my model didn't do horrible, stock price-predicting Neural Networks have potential to achieve exponential better accuracy than mine did. This would explain why many investment firms and banks keep their models as trade secrets and away from the public.

Outside of the region of the stock market, however, Neural Networks have fundamentally countless applications from speech recognition to potentially predicting quantum properties of molecules. There is much more research being done with machine learning because the applications for these models have been very promising and exciting. Advancements and innovation continue each day, and it may only be a matter of time until artificial intelligence is as efficient as our own.

Acknowledgements
The author thanks Mr. Robert Gotwals for providing a very nice LaTeX template. Appreciation is also extended Siraj Raval, who helped inspire me to write about this topic and to the YouTube channel "Sentdex", for acquainting me with python and TensorFlow.

References

[1] History of coins in Italy, January 2018. Page Version ID: 818870619.

[2] Hayes Adam. Stock Basics: What are Stocks?, May 2017.

[3] Rick Ferri. Any Monkey Can Beat The Stock Market. *Forbes Media LLC.*, December 2012.

[4] K.P. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive computation and machine learning. MIT Press, 2012.

[5] Neural Networks - Sophomore College 2000.

[6] Multilayer perceptron, September 2017. Page Version ID: 798309764.

[7] Michael A. Nielsen. Neural Networks and Deep Learning. 2015.

[8] Understanding LSTM Networks – colah's blog.

1 Supplemental Code pt.1: Getting the Stock Data

```
#using yahoo api to get stock data and convert to .csv

#imports
import pandas as pd
import pandas_datareader.data as web
import datetime as dt

#setting start and end dates for the data we wanna call
start = dt.datetime(2016, 1, 1)
```

```
end = dt.datetime(2016, 12, 31)

#Calling in the data
df =  web.DataReader('TSLA', 'yahoo', start, end)

#Writing to csv
df.to_csv('TSLA.csv')
```

2 Supplemental Code pt.2: Parsing the Data

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.dates as mdates

#Read data
data = pd.read_csv('TSLA.csv', parse_dates=True, index_col=0)
prices  = data['Adj Close']
n_rows = prices.shape[0]
n_columns = 1
prices = prices.values
dates = data.index
dates2 = dates.values


if __name__ == "__main__":
    #the plot
    plt.figure(figsize=(10,5))
    plt.title('TSLA Stocks')
    plt.ylabel('Price: $$$')
    plt.xlabel('Dates')
    plt.plot(dates, prices, '-b', label = 'Adjusted Closing Price')
    #plt.plot(dates, prices2, '-r', label= 'Bye')
    plt.legend(loc='upper right')
    plt.savefig('TSLA_Stocks.png')
    #plt.scatter(prices, prices2)
    #plt.show()
```

3 Supplemental Code pt.3: Formatting the Data for the nueral Network

```
from dataFormat import *
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt

# Scaling
scale = [i for i in range(n_rows)]
data = np.transpose([scale, prices])
scaler = MinMaxScaler()
scaler.fit(data)
data = scaler.transform(data)
data = np.transpose(data)
prices_scaled = data[1]
dates_scaled = data[0]
#code for un-scaling data
    #inversed = scaler.inverse_transform(np.transpose(data))
    #inversed = np.transpose(inversed)

#training data 0-216, testing data 217-252
train_y = prices_scaled[:216]
test_y = prices_scaled[216:]
train_x = dates_scaled[:216]
test_x= dates_scaled[216:]
dates_train = dates2[:216]
dates_test= dates2[216:]
dates_model = pd.DatetimeIndex(dates_test)
test_prices = prices[216:]

#reshaping the data
train_y = train_y.reshape((6, 1, 36))
train_x = train_x.reshape((6, 1, 36))
test_y = test_y.reshape((1,36))
test_x = test_x.reshape((1, 36))

if __name__ == "__main__":
    #the plot
```

```
    plt.figure(figsize=(10,5))
    plt.title('TSLA Stocks (Scaled Testing Data)')
    plt.ylabel('Scaled Price')
    plt.xlabel('Scaled Dates')
    plt.plot(test_x, test_y, '-b', label = 'Adjusted Closing Price')
    plt.legend(loc='upper right')
    plt.savefig('TSLA_Stocks_Scaled_Test.png')
```

4 Supplemental Code pt.4: The RNN

```
#imports
from data_prep import *
import TensorFlow as tf
from TensorFlow.contrib import rnn
#Reshaping the data-- input: (6, 36) labels: (6, 36)
n_epochs = 50
rnn_size = 198
n_seqs = 6
timestep =6
batch_size = 6
n_outputs = 36

#PlaceHolders
x = tf.placeholder('float', shape=[None,  n_seqs])
y = tf.placeholder('float')


def computational_graph(x):

    layer = {'weights':tf.Variable(tf.random_normal([rnn_size, n_outputs])),
             'biases' :tf.Variable(tf.random_normal([n_outputs]))}

    x = tf.reshape(x, [-1, n_seqs])
    x = tf.split(x, timestep, 0)

    lstm_cell = rnn.BasicLSTMCell(rnn_size)
    outputs, states = rnn.static_rnn(lstm_cell, x, dtype=tf.float32)
    output = tf.add(tf.matmul(outputs[-1], layer['weights']), layer['biases'])
```

```
    print(output)

    return output


def train_graph(x):
    output = computational_graph(x)

    costFunc = tf.reduce_mean( tf.squared_difference(output, y))
    optFunc = tf.train.AdamOptimizer(learning_rate=0.003).minimize(costFunc)

    #starting tf session
    with tf.Session() as ses:
        #intitializing all tf variables
        ses.run(tf.global_variables_initializer())
        #lists for plotting the loss and price action
        loss_list = []
        accuracies = []

        for epoch in range(n_epochs):

                #batch training
                for i in range(0, batch_size-1):
                    #total loss is added conseuctively per epoch
                    epoch_loss = 0
                    batch_x = train_x.reshape((batch_size, n_seqs, timestep))
                    batch_y = train_y
                    start = i * batch_size
                    batch_x = batch_x[i]
                    batch_y = batch_y[i]
                    # Run optimizer for each batch
                    i, c = ses.run([optFunc, costFunc], feed_dict={x: batch_x, y
                    epoch_loss += c
                    loss_list.append(epoch_loss)


                    print('Epoch-Loss-', str(epoch), '-of-',str(n_epochs),'-',st

                    # retreiving the predictions
                    pred = ses.run(output, feed_dict={x: test_x.reshape((n_seqs,
                    pred = pred.tolist()
```

```
#print(pred)

#un-scaling predicted data into real data
plot_x = test_x.tolist()
holder1 = []
holder2 = []
for w in pred:
    for z in w:
        holder1.append(z)
for w in plot_x:
    for z in w:
        holder2.append(z)
plot_data = np.transpose([holder2, holder1])
inversed = scaler.inverse_transform(plot_data)
inversed = np.transpose(inversed)
inversed = list(reversed(inversed[1]))

#making the  stock figure
plt.figure(figsize=(10,5))
plt.title('TSLA Stocks Prediction')
plt.ylabel('Price: $$$')
plt.xlabel('Dates')
plt.plot(dates_model, test_prices, '-b', label = 'Adjusted C
plt.plot(dates_model, inversed, '-r', label = 'Adjusted Clos
plt.legend(loc='upper right')
plt.grid()
file_name = 'epoch_graph2/epoch_' + str(epoch) +  '.jpg'
plt.savefig(file_name)
plt.clf()

#making scatterplot
plt.figure(figsize=(5,5))
plt.title('TSLA Predicted vs. Actual Stocks')
plt.ylabel('Predicted Price')
plt.xlabel('Actual Price')
plt.scatter(test_prices, inversed, label = 'Adjusted Closing
plt.grid()
file_name2 = 'epoch_graph2/Scatter_epoch_' + str(epoch) +  '
plt.savefig(file_name2)
plt.clf()
```

```
        #making price-action accuracy function

        a = test_prices
        b = inversed
        e = [i for i in range(1, 51)]
        accuracy = 0
        for i in range(0, (len(a)-1)):

            if (a[i] - a[i+1] < 0 and b[i] - b[i+1] < 0) or (a[i] - a[i+
                    accuracy += 1

        z = 100 * accuracy/(len(a)-1)
        accuracies.append(z)
        print('Epcoh', epoch, 'Accuracy: ', z, '%')

#making error plot
current_epoch = [((i*(1/(batch_size-1)))) for i in range(0,int(((batch_s
plt.figure(figsize=(5,5))
plt.title('Mean Squared Error')
plt.ylabel('Error Value')
plt.xlabel('Epoch number')
plt.plot(current_epoch, loss_list)
plt.grid()
file_name3 = 'errorGraph.jpg'
plt.savefig(file_name3)
plt.clf()

#price-wise accruacy plot
plt.figure(figsize=(5,5))
plt.title('Price Action Accuracy')
plt.ylabel('Percentage Correct %')
plt.xlabel('Epoch number')
plt.bar(e, accuracies, width=0.5, color='green', edgecolor='black')
plt.yticks([0,10,20,30,40,50,60,70,80,90,100])
plt.grid()
file_name4 = 'accuracyGraph.jpg'
plt.savefig(file_name4)
plt.clf()
```

```
train_graph(x)
```

5 figures:

Figure 2: This figure shows the entire year's stock prices for TSLA

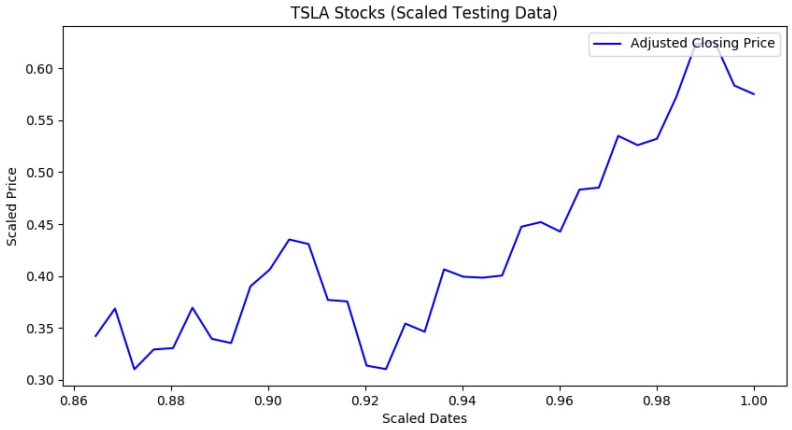

Figure 3: This figure shows the actual prices and dates for the testing data which have been scaled

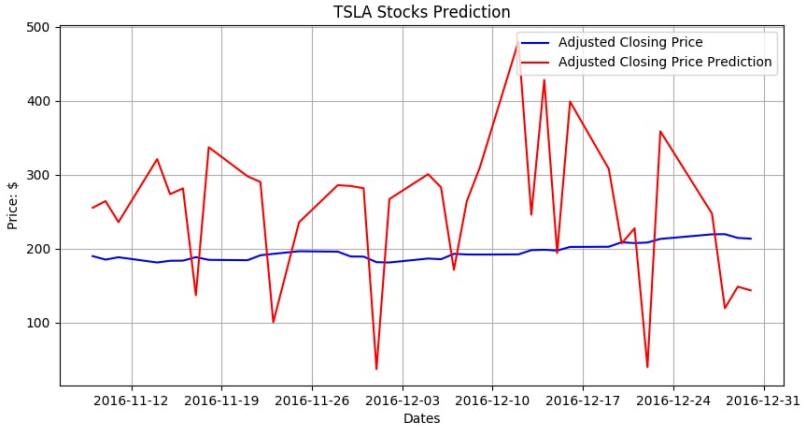

Figure 4: This figure shows the RNN's stock price predictions after 1 epoch.

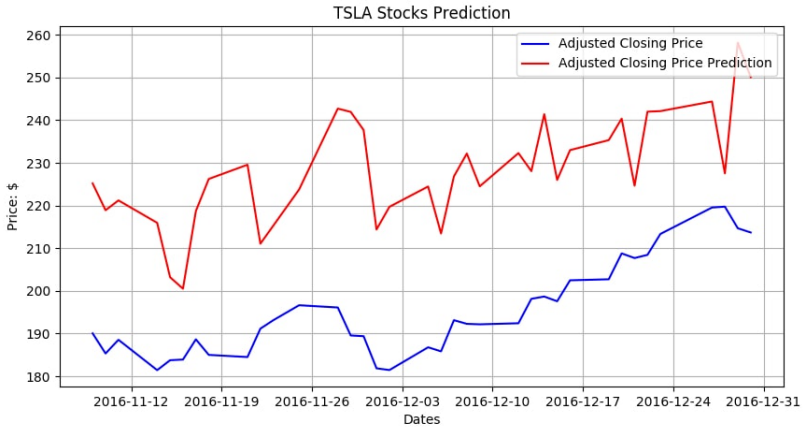

Figure 5: This figure shows the RNN's stock price predictions after 30 epochs.

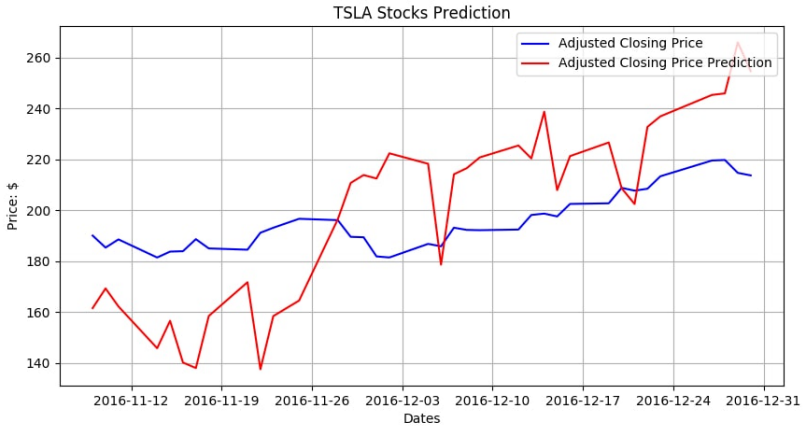

Figure 6: This figure shows the RNN's stock price predictions after 50 epochs.

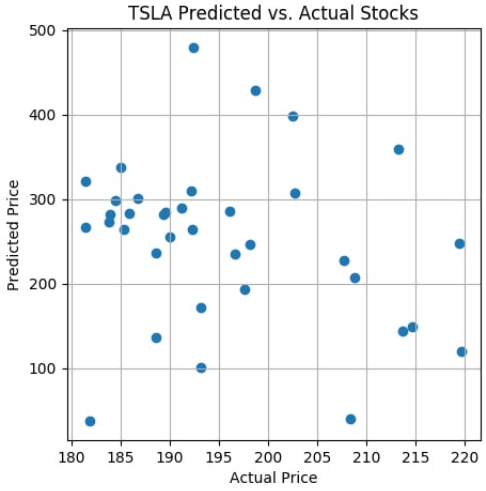

Figure 7: This figure plots the price predictions against the actual prices after 1 epoch.

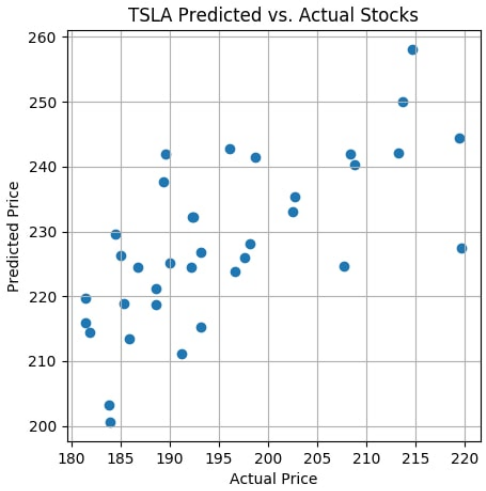

Figure 8: This figure plots the price predictions against the actual prices after 30 epochs.

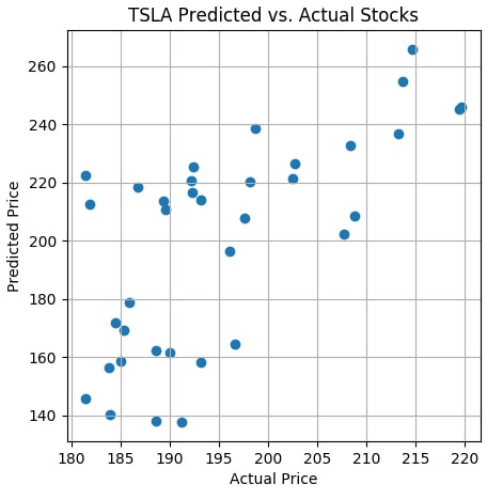

Figure 9: This figure plots the price predictions against the actual prices after 50 epochs.

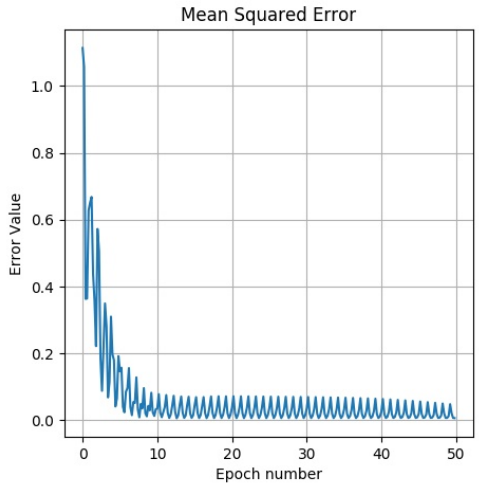

Figure 10: This figure shows the over-time MSE for each 50 epochs.

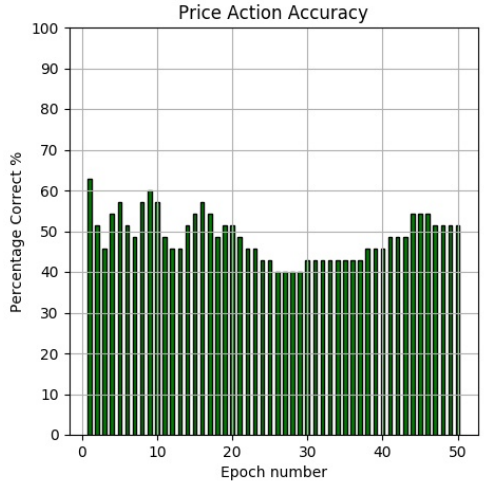

Figure 11: This figure shows the percentage of the network's predictions moving in the same direction as the actual prices during a 36 day period for each epoch.